AD-A216 747

# REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188



| 1. AGENCY USE ONLY (Leave blank) | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | | FINAL 1 Oct. 87 thru 30 Sep 88 |

**4. TITLE AND SUBTITLE**
HIGH PERFORMANCE COMPUTER PROGRAMMING ENVIRONMENTS

**5. FUNDING NUMBERS**
(2)

**6. AUTHOR(S)**
Lawrence Snyder
David Notkin

DTIC
ELECTE
JAN 16 1990
S D
D CS

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
University of Washington
Computer Science Department
Seattle, WA 98195

**8. PERFORMING ORGANIZATION REPORT NUMBER**
G- AFOSR-83-0023
PE- 61102F
PR-2304 TA-A2

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
AIR FORCE OFFICE OF SCIENTIFIC RESEARCH
Mathematical and Information Sciences
Building 410
Bolling AFB, DC 20332-6448
AFOSR

**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
TR. 89-1682

**11. SUPPLEMENTARY NOTES**

**12a. DISTRIBUTION/AVAILABILITY STATEMENT**
Approved for public release; Distribution unlimited.

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** *(Maximum 200 words)*

This one year grant had the primary goal the assessment of the Poker Parallel Programming environment and the planning and design of new parallel programming environment. These goals were achieved.The new programming environment, to be built on a software platform that permits rapid prototyping of alternative environments, was designed using a three level language abstraction. The central publications include the assessment of Poker, two papers on prototype graphic debugging environment, and two papers on parallel computer structures. Thomas J. Holman completed his Ph.D. degree

90 01 11 127

| 14. SUBJECT TERMS | 15. NUMBER OF PAGES |
|---|---|
| | 9 |
| | 16. PRICE CODE |

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | SAR |

# Final Report

**Grant Number**: AFOSR-88-0023

**Title**: High Performance Computer Programming Environments

**Principal Investigators**: Lawrence Snyder and David Notkin

**Period**: 1 October 1987 - 30 September 1988

## 1 Executive Summary.

This one year grant had as its primary goal the assessment of the Poker Parallel Programming Environment [Snyder 84] and the planning and design of a new parallel programming environment. These goals were achieved. The new programming environment, to be built on a software platform that permits rapid prototyping of alternative environments, was designed using a three level language abstraction [Snyder 89]. The central publications included the assessment of Poker [Notkin *et al.* 88], two papers on a prototype graphic debugging environment, *Voyeur* [Bailey, Socha & Notkin 88, Socha, Bailey & Notkin 88] and two papers on parallel computer structures [Snyder 88, Holman & Snyder 89]. The Poker Parallel Programming environment was distributed under the aegis of this grant [Poker 88], permitting other researchers to assess it. Thomas J. Holman completed his Ph. D. degree.

## 2 Introduction and Background.

The proposal setting forth the plans for this project called for a five year effort covering the design, development, implementation, testing, assessment and distribution of a new parallel programming environment founded on the concepts pioneered in the Poker Parallel Programming Environment [Snyder 84]. The plan, enunciated by Capt. John Thomas of AFOSR, was to begin with an initial award of three years at the budget levels presented in the revised budget; with the understanding that satisfactory performance would result in funding for the final two years. The funding was terminated after the first year by Dr. Abraham Waksman of AFOSR for reasons not related to our performance on the grant. This report covers the work conducted during that one year.

The proposed programming environment, eventually named the Orca, has its intellectual roots in the Poker Parallel Programming Environment. Developed over the period 1982–1988

with ONR and NSF funding, Poker was the first parallel programming environment and the first programming language to demonstrate portability across different parallel machines. It provides graphic programming facilities for the challenging, but most general, model of parallel computation, the nonshared or distributed memory model. Because Poker sought to introduce radically new ideas about parallel programming and because most of the Poker team was still intact when the present project began, it was natural for the research to begin by evaluating the Poker experiment. That work [Notkin *et al.* 88] is described in the next section.

Once Poker had been assessed, the planning for the new environment began. In an effort to understand how the difficult task of parallel program debugging might be more effectively supported than it was with the Poker environment, a prototype graphic debugger was developed. The debugger was called Voyeur [Bailey, Socha & Notkin 88, Socha, Bailey & Notkin 88]. Though Voyeur was interfaced to the Poker environment, the intent was to develop a facility that would be suitable for the more ambitious goals of Orca. Voyeur demonstrated its effectiveness by finding errors in a variety of Poker programs and its development continues. Voyeur is described in Section 4.

The planning and design of the Orca environment progressed throughout the entire year. Implementation of a prototype system began after the conclusion of the current grant.

An important characteristic of parallel software that is not shared by software for sequential machines is a greater dependence on the architectural characteristics of the underlying execution engine; architecture affects parallel software more. Accordingly, it is natural to conduct architectural research concurrently with parallel software development, and this policy lead to publication of two papers on parallel architecture design [Snyder 88, Holman & Snyder 89]. The first produced the first taxonomy of synchronous parallel machines capable of explaining *why* they must be synchronous. The second provided a new design methodology for parallel computers wherein the components included in a machine have to "pay their way" in speeding up the execution of (Poker) programs when compared against using the hardware for more processing elements of a simpler kind. Both papers generated considerable interest in the architecture community and they are described in Sections 5 and 6, respectively.

# 3 Assessment of Poker

Poker was an experimental system with two goals — to be a useful programming tool, and to create new ways of programming parallel computers. To meet the latter objective, many nontraditional language mechanisms were included in the environment. Like any instance where many new ideas are tried, we found that some worked and some did not. Before building the new environment, we assessed each Poker mechanism to determine which should be retained [Notkin *et al.* 88].

Recall that Poker [Snyder 84] provides an integrated programming environment to support the distributed memory model of parallel computation; see Figure 1. It uses interactive graphics to show the programmer metaphorically rich pictures of a programming situation

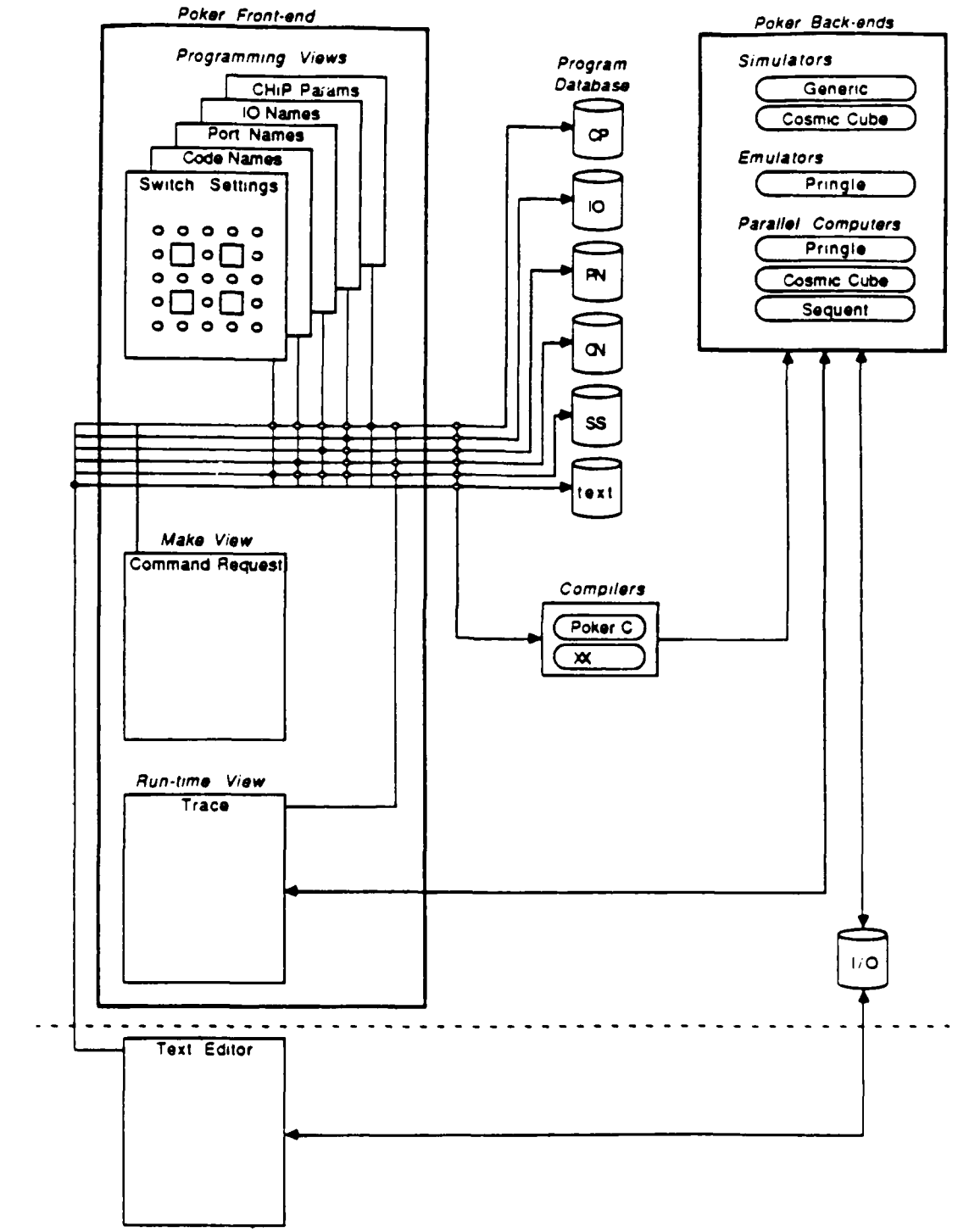


Figure 1: The Structure of Poker

| | |
|---|---|
| Cholesky decomposition | FFT |
| Dynamic programming | SIMPLE |
| Matrix multiply (systolic) | ADI |
| Band matrix multiply | SOR |
| (systolic) | Polygon clipping |
| Vector-matrix multiply | WAP (systolic) |
| Matrix multiply | LU-decomposition |
| (divide & conquer) | Transitive closure |
| Topological sort | Batcher's sort |
| Conjugate gradient | Jacobi iteration |
| Sharks & fishes | Game of Life |
| Dataflow simulator | |

Table 1: Programs Built Using Poker

customized to his particular problem. The environment executes on a scientific workstation and cross compiles to the parallel machine. Since it does not represent programs as monolithic pieces of code, it stores the program in a database. Poker uses an extended version of C as the process language, it has interactive debugging facilities (apart from Voyeur), and provides library and operating system interfaces.

Based on the experience derived from writing the set of programs listed in Table 1, the principal findings of the Poker evaluation can be summarized informally as follows:

- Interactive graphics is a power tool for simplifying parallel programming.
- The nonsymbolic, database representation of the program (from which synthetic pictures of components of the program can be generated) provides the basis for a wide range of programmer support with considerable research potential.
- The relational database of Poker was difficult to use and an object oriented approach is likely to work better.
- A more powerful high level (technically a "Z level") language is needed: Poker's was too weak.
- The explicit presentation of the communication structure of the algorithm is crucial to Poker's retargetability and efficiency.
- Poker's program form was too rigid.
- Though Poker is one of few languages that treats I/O as something other than an afterthought, it was too low level.
- Poker's use of simulators and its interactive, graphic debugging are a good beginning step for supporting the difficult task of parallel program debugging; performance debugging are also crucial. Though many things can and have been criticized about

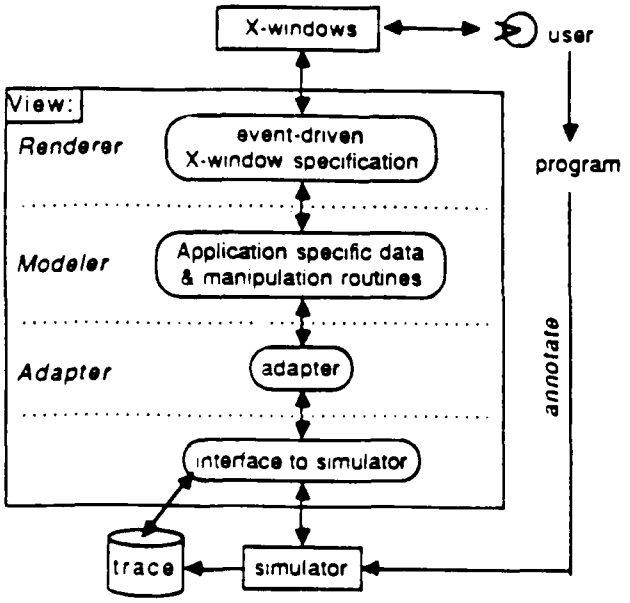


Figure 2: View System Structure

Poker, it was a successful experiment. Many of the ideas tried in Poker are only now being proposed for other languages. As a result, the new environment design begins on a stable foundation of experience.

# 4 Voyeur

Voyeur is a prototype system for viewing and graphically debugging nonshared memory parallel programs [Bailey, Socha & Notkin 88, Socha, Bailey & Notkin 88] . The motivation for Voyeur begins with two facts: (1) Poker has a generic graphic debugging facility which at the time of its development was superior to any other parallel debugging facility and (2) debugging parallel programs is sufficiently difficult that more assistance is needed by the programmer. The goal for Voyeur then was to move beyond what was provided as standard in Poker. The result is a convenient facility for programmer-customized graphic illustrations of programs, which can give filtered and interpreted information about their execution. The resulting displays have high information content leading to rapid discovery of bugs.

The basic logic of Voyeur is given in Figure 2. The system is event driven, capable of processing events from an annotated program running in "realtime" in a Poker simulator, or processing events from a postmortem trace.

There are three primary components: an adapter, a modeler and a renderer. The adapter serves simply as an interface between the different formats produced by the annotated programs and the formats needed within Voyeur. The modeler manages application-specific

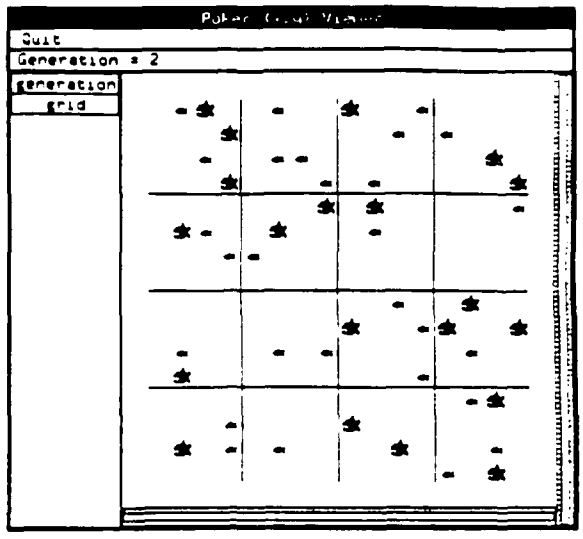


Figure 3: An x, y plot of shark and fish icons

data, i.e. it keeps the internal representation of what is to be displayed to the user. The renderer produces the synthetic picture that the user sees on the screen. The X windows system provides the system support for the displays.

Voyeur was used to produce a trace view similar to that originally provided by Poker. This is still a generic facility. To customize displays for application-specific debugging, support for other views was provided: The icon view enabled programs like the Sharks and Fishes (an ecology simulation) to be displayed; see Figure 3. The distributed linked list view permitted program data structures to be displayed. The vector view, which was used to show the runtime behavior of the Simple program, provided an easy but powerful way of directly illustrating program values.

The power of the program-specific debugging was convincingly illustrated with Voyeur. Three bugs were quickly found in the Sharks and Fishes program. For the Simple program a bug was *not* found. In particular, using the Voyeur vector view, the anomolous behavior of the program was shown to be due to numerical instability rather than a bug in the algorithm.

Following the completion of the contract, work has continued on Voyeur. Color has been added, the system has been reprogrammed for the C++ language, and support has been provided in the new environment for Voyeur-type debugging.

# 5 Synchronous Parallel Computation

Though a large number of computer taxonomies have been developed over the last several decades, all have been inadequate. Many have been overly influenced by technology, and nearly all focus on internal physical rather than logical structure. Flynn's taxonomy, which

gave us the SIMD and MIMD designations, is undoubtedly the most famous and widely used, but it has only four groups and one of those is dubious (MISD). A small contribution to the taxonomization of computers has been developed [Snyder 88].

The key contribution of this work is a classification of the synchronous computers which carries with it an explanation of *why* they must be synchronous. This is accomplished by following Flynn's lead in recognizing the importance of the instruction stream (I- stream) and the data stream (D-stream) of computers, but departing from Flynn by recognizing that these streams are not monolithic. They are formed of addresses, generally sent from the processors to the memories, and values, which can move both ways between processors and memories. The values are interpreted differently depending upon which stream they are a part of — either instructions or data. After some manipulation and analysis, machines can be classified based on the number of components in their I- and D-streams.

Though the full taxonomic development is needed before classifications of specific machines can be presented, it is possible to give a hint of why synchronous machines are synchronous. Basically, any machine that has a single element in the address component of its I-stream must be synchronous no matter how complex the remainder of operations are. This is because the establishment of that address (usually by incrementing an instruction counter) will serve as the synchronizing point. All other times can be defined relative from the address time, all other operations must complete before the next address time begins, etc. If there are multiple instruction addresses being generated they can skew in time and become asynchronous, but one address has nothing to skew with respect to.

A substantial amount of additional taxonomic work has been completed but remained unpublished at the conclusion of the grant.

# 6 Profiles of Programs

Programming environment research of the kind supported by this grant is generally expected to produce results in software, but the work to be described in this section is hardware research that directly benefitted from our study of programming environments. Before explaining how, we must explain what we did [Holman & Snyder 89].

When building parallel computers there is always a tradeoff between using the hardware for a modest number of extremely fast machines or a larger number of machines that are not maximally fast. With a little thought it is clear that no absolute choice for this is possible. Any answer depends on two "variable" phenomena — how much of the current hardware technology is needed to implement any particular circuit, and which facilities are most used in the programs users want to run. Assuming we had specific numbers for these variable phenomena, we have developed a methodology to answer the "tradeoff" question — a few fast processors *vs* many slower processors. Current VLSI technology gives us hard numbers for the first of the variable phenomena. An experimental simulator for the programming environments and programs from Table 1 provide the numbers for the second variable.

The simulator in question was developed for Poker to test out the idea that when getting a parallel program functionally correct, the specific details of the parallel hardware are

unimportant. Thus a "generic" parallel machine would be sufficient, and could run faster. So we developed a simulator which provided an idealized machine environment for functional debugging. One of the features of the simulator is its ability to fully instrument a program, counting the dynamic frequency of all of the instructions executed and the time spent in communication and waiting. This provides exactly the data needed to evaluate the tradeoffs mentioned above.

The methodology has been used to analyze the structures architects use to make parallel computer processing elements fast. We have validated quantitatively the conventional wisdom that floating point units are cost-performant. Specifically, we have shown that if the transistors used for the floating point units of Transputer T-800 chips were used to make more processors and the floating point instructions were simulated in software (the way they were on the T414) then the machine would be slower. Conversely, the conventional wisdom that says that barrel shifters are a good addition to ALUs is wrong by this analysis. It would be better to use the transistors used in barrel shifters for more processors and to simulate the shifts in software. A variety of other features were also studied.

# 7 Conclusions.

The planning and design of a new parallel programming environment were undertaken. Along the way a number of publications and software were produced. These include an evaluation of the novel features of Poker, the design and implementation of a new graphic debugger, Voyeur, and a study involving the impact of the dynamic behavior of parallel programs on parallel machine design.

## References


[Bailey, Socha & Notkin 88]
M. Bailey, D. Socha and D. Notkin
Parallel Debugging Using Graphical Views
*Proceedings of the International Conference on Parallel Processing,*
Penn State, pp. 46–49

[Holman & Snyder 89]
T.J. Holman and L. Snyder
Architectural Tradeoffs in Parallel Computer Design
In Charles L. Seitz (ed.) *Proceedings Decennial Caltech Conference on VLSI,*
pp. 317–334, 1989

[Notkin, *el al.* 88]
D. Notkin, D. Socha, L. Snyder, M. Bailey, B. Forstall, K. Gates, R. Greenlaw, W. Griswold, T. Holman, R. Korry, G. Lasswell, R. Mitchell and P. Nelson
Experiences with Poker
*Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages and Systems,* July 1988, pp. 10–20

[Poker 88]
Poker (4.2) Reference Manual
University of Washington, Technical Report 88-10-05, 1988

[Snyder 84]
Lawrence Snyder
Parallel Programming and the Poker Programming Environment
*IEEE Computer* 17(7):pp. 27–36, July, 1984

[Snyder 88]
Lawrence Snyder
A Taxonomy of Snychronous Parallel Machines
*Proceedings of the International Conference on Parallel Processing*, pp. 281–285

[Snyder 89]
Lawrence Snyder
The XYZ Abstraction Levels of Poker-like Languages
Springer-Verlag LNCS Series, (to appear)

[Socha, Bailey & Notkin 88]
D. Socha, M. Bailey and D. Notkin
Voyeur: Graphical Views of Parallel Programs
*Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, ACM pp. 206–215